CS420 – Lecture 11

Marc Snir

Fall 2018

Quiz

What is the maximum length vector operation which can be applied to the following loop? Assume the largest vector register in 512 bits, an int is 32 bits, and a and b have N elements. Give your answer in terms of the number of ints that a vector operation can be applied to.

```
int *a, *b;
int N = 10000;
...
for (int i = 16; i < N; ++i) {
  a[i] = a[i-4] + b[i-10];
}
```

Quiz

What is the maximum length vector operation which can be applied to the following loop? Assume the largest vector register in 512 bits, an int is 32 bits, and a and b have N elements. Give your answer in terms of the number of ints that a vector operation can be applied to.

```
int *a, *b;
int N = 10000;
...
for (int i = 16; i < N; ++i) {
  a[i] = a[i-4] + b[i-10];
}
```

Naive answer: 4

Quiz

What is the maximum length vector operation which can be applied to the following loop? Assume the largest vector register in 512 bits, an int is 32 bits, and a and b have N elements. Give your answer in terms of the number of ints that a vector operation can be applied to.

```
int *a, *b;
int N = 10000;
...
for (int i = 16; i < N; ++i) {
  a[i] = a[i-4] + b[i-10];
}
```

Naive answer: 4
Correct answer: 1
a and b can be aliased.

```
#include <stdlib.h>
 int main() {
  int *a, *b;
  int N = 10000;
  a= (int*)malloc(1100*sizeof(int));
  b = a+9;
  for (int i = 16; i < N; ++i) {
    a[i] = a[i-4] + b[i-10];
  }
}
```

b[i-10] is the same as a[i-1].

Enabling vectorization

```
int *restrict a;
int *restrict b;
int N = 10000;
...
for (int i = 16; i < N; ++i) {
  a[i] = a[i-4] + b[i-10];
}
```

I promise that locations accessed through `a` are not accessed through another pointer (and same for `b`).
Compiler can vectorize (to length 4) based on this promise. If, nevertheless, `a` and `b` are aliased, the code has a bug and the result is undefined.

What is spatial data locality?
Answer

- If data at a particular address have been used recently, then that data will be used again soon.
- If data at a particular address have been used recently, then data at nearby addresses will be used soon.
- If data at a particular address have been used recently, then then no data within a given distance of that address will be used for a given amount of time.
- At any time, data at all addresses have equal probability of being used.

What is spatial data locality?
Answer

- If data at a particular address have been used recently, then that data will be used again soon.
- If data at a particular address have been used recently, then data at nearby addresses will be used soon.
- If data at a particular address have been used recently, then then no data within a given distance of that address will be used for a given amount of time.
- At any time, data at all addresses have equal probability of being used.

Question 3 and 4 are based on this code segment.

```
struct pair {
  int x;
  int y;
};

static struct pair f;

int sum_x(void) {
  int s = 0;
  int i;
  for (i = 0; i < 1000000; ++i)
  s += f.x;
  return s;
}

void inc_y(void) {
  int i;
  for (i = 0; i < 1000000; ++i)
  ++f.y;
}
```

When the functions sum_x and inc_y are running concurrently by two threads on this multicore machine, what type of cache misses could happen?

Answer

- true sharing misses only
- false sharing misses only
- true sharing misses and false sharing misses
- no cache misses

Question 3 and 4 are based on this code segment.

```
struct pair {
  int x;
  int y;
};

static struct pair f;

int sum_x(void) {
  int s = 0;
  int i;
  for (i = 0; i < 1000000; ++i)
  s += f.x;
  return s;
}

void inc_y(void) {
  int i;
  for (i = 0; i < 1000000; ++i)
  ++f.y;
}
```

When the functions sum_x and inc_y are running concurrently by two threads on this multicore machine, what type of cache misses could happen?

Answer

- true sharing misses only
- false sharing misses only
- true sharing misses and false sharing misses
- no cache misses

Question 3 and 4 are based on this code segment.

```
struct pair {
  int x;
  int y;
};

static struct pair f;

int sum_x(void) {
  int s = 0;
  int i;
  for (i = 0; i < 1000000; ++i)
  s += f.x;
  return s;
}

void inc_y(void) {
  int i;
  for (i = 0; i < 1000000; ++i)
  ++f.y;
}
```

If L1 cache design is changed to hold only a one-word data block in each line, when the functions sum_x and inc_y are running concurrently on this multicore machine, what type of misses can happen?

Answer

- true sharing misses only
- false sharing misses only
- true sharing and false sharing misses
- no cache misses

Question 3 and 4 are based on this code segment.

```
struct pair {
  int x;
  int y;
};

static struct pair f;

int sum_x(void) {
  int s = 0;
  int i;
  for (i = 0; i < 1000000; ++i)
  s += f.x;
  return s;
}

void inc_y(void) {
  int i;
  for (i = 0; i < 1000000; ++i)
  ++f.y;
}
```

If L1 cache design is changed to hold only a one-word data block in each line, when the functions sum_x and inc_y are running concurrently on this multicore machine, what type of misses can happen?

Answer

- true sharing misses only
- false sharing misses only
- true sharing and false sharing misses
- no cache misses

Graph traversal Possible Improvements

- All processes first send to process 0 then to process 1, etc.;
- Communication not overlapped optimally
- Better: Process i sends to $i+1, i+2, \ldots$ and receives from $i-1, i-2, \ldots$

```
k=0;
m = rank;
for(i=0;i<size-1;i++) {
 m=(m+1)%size;
MPI_Isend(out[m],out_ptr[m], MPI_INT, m, 0, MPI_COMM_WORLD, &reqs[k++]);
```

- Use multiple receive buffers, so that all receives can be started at the same time
- But, then may need to allocate more space
- Check how much space each send needs, then allocate receive buffer

```
k=0;
m = rank;
for(i=0;i<size-1;i++) {
 m=(m-1)%size;
 MPI_Probe(m,0,MPI_COMM_WORLD,status);
 MPI_Get_count(&status, MPI_INT,&j);
 MPI_Irecv(in[in_ptr], j, MPI_INT, m, 0, MPI_COMM_WORLD, &reqs[k++]);
 int_ptr += j;
}
```

- `MPI_Probe(source, tag, comm, status)`: Returns information about incoming message without actually receiving it

Some examples of incorrect codes

Assume there are 2 processes

Some examples of incorrect codes

Assume there are 2 processes

```
if(myrank==0)
 MPI_Isend(a,100,MPI_INT,1,0,MPI_COMM_WORLD,&req);
else
 MPI_Irecv(a,100,MPI_INT,0,0,MPI_COMM_WORLD,&req);
a[3]=5;
MPI_Wait(&req,MPI_STATUS_IGNORE);
```

Some examples of incorrect codes

Assume there are 2 processes

```
if(myrank==0)
 MPI_Isend(a,100,MPI_INT,1,0,MPI_COMM_WORLD,&req);
else
 MPI_Irecv(a,100,MPI_INT,0,0,MPI_COMM_WORLD,&req);
a[3]=5;
MPI_Wait(&req,MPI_STATUS_IGNORE);
```

Race

Some examples of incorrect codes

Assume there are 2 processes

```
if(myrank==0)
 MPI_Isend(a,100,MPI_INT,1,0,MPI_COMM_WORLD,&req);
else
 MPI_Irecv(a,100,MPI_INT,0,0,MPI_COMM_WORLD,&req);
a[3]=5;
MPI_Wait(&req,MPI_STATUS_IGNORE);
```

Race

```
if(myrank==0)
 MPI_Isend(a,100,MPI_INT,1,0,MPI_COMM_WORLD,&req);
else /* myrank==1 */
MPI_Irecv(a,100,MPI_INT,0,0,MPI_COMM_WORLD,&req);
b=a[3];
MPI_Wait(&req,MPI_STATUS_IGNORE);
```

Some examples of incorrect codes

Assume there are 2 processes

```
if(myrank==0)
 MPI_Isend(a,100,MPI_INT,1,0,MPI_COMM_WORLD,&req);
else
 MPI_Irecv(a,100,MPI_INT,0,0,MPI_COMM_WORLD,&req);
a[3]=5;
MPI_Wait(&req,MPI_STATUS_IGNORE);
```

Race

```
if(myrank==0)
 MPI_Isend(a,100,MPI_INT,1,0,MPI_COMM_WORLD,&req);
else /* myrank==1 */
MPI_Irecv(a,100,MPI_INT,0,0,MPI_COMM_WORLD,&req);
b=a[3];
MPI_Wait(&req,MPI_STATUS_IGNORE);
```

Race, again

Cont.

```
if(myrank==0) {
 MPI_Send(a,100,MPI_INT,1,0,MPI_COMM_WORLD);
 MPI_Reduce(b,c,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
 }
else { /* myrank==1 */
 MPI_Reduce(b,c,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
 MPI_Recv(a,100,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
 }
```

Cont.

```
if(myrank==0) {
 MPI_Send(a,100,MPI_INT,1,0,MPI_COMM_WORLD);
 MPI_Reduce(b,c,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
 }
else { /* myrank==1 */
 MPI_Reduce(b,c,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
 MPI_Recv(a,100,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
 }
```

Deadlock: A collective operation may synchronize the processes involved: It could be that none can continue beyond the collective call until all have started executing it. (On the other hand, it could be that some complete the collective call before all have started it.)

More MPI quizzes

```
int main() {
    MPI_Init(NULL, NULL);
    int myrank;
    char a,b,c,d;
    a='a'; b='b';
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank==0) {
        MPI_Send(&a, 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
        MPI_Send(&b, 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    }
    else {
        MPI_Recv(&c, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
        MPI_Recv(&d, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
        printf("%c, %c \n", c, d);
    }
    MPI_Finalize();
}
```

More MPI quizzes

```
int main() {
    MPI_Init(NULL, NULL);
    int myrank;
    char a,b,c,d;
    a='a'; b='b';
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank==0) {
        MPI_Send(&a, 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
        MPI_Send(&b, 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    }
    else {
        MPI_Recv(&c, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&d, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("%c, %c \n", c, d);
    }
    MPI_Finalize();
}
```

Prints a, b

More MPI quizzes

```
int main() {
    MPI_Init(NULL, NULL);
    int myrank;
    char a,b,c,d;
    a='a'; b='b';
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank==0) {
        MPI_Send(&a, 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
        MPI_Send(&b, 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    }
    else {
        MPI_Recv(&c, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&d, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("%c, %c \n", c, d);
    }
    MPI_Finalize();
}
```

Prints a, b

Messages are received in the order they were sent

```
#include<stdio.h>
#include<mpi.h>
int main() {
    MPI_Init(NULL, NULL);
    int myrank;
    char a,b,c,d;
    a='a'; b='b';
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank==0) {
        MPI_Send(&a, 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
        MPI_Send(&b, 1, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
    }
    else {
        MPI_Recv(&c, 1, MPI_CHAR, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&d, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("%c, %c \n", c, d);
    }
    MPI_Finalize();
}
```

```
#include<stdio.h>
#include<mpi.h>
int main() {
    MPI_Init(NULL, NULL);
    int myrank;
    char a,b,c,d;
    a='a'; b='b';
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank==0) {
        MPI_Send(&a, 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
        MPI_Send(&b, 1, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
    }
    else {
        MPI_Recv(&c, 1, MPI_CHAR, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&d, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("%c, %c \n", c, d);
    }
    MPI_Finalize();
}
```

Either prints b, a or deadlocks

```
...
int myrank;
char a,b,c,d;
MPI_Request req[2];
a='a'; b='b';
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank==0) {
MPI_Isend(&a, 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD,&req[0]);
MPI_Isend(&b, 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &req[1]);
MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
}
else {
MPI_Recv(&c, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Recv(&d, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
printf("%c, %c \n", c, d);
}
...
```

```
...
int myrank;
char a,b,c,d;
MPI_Request req[2];
a='a'; b='b';
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank==0) {
MPI_Isend(&a, 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD,&req[0]);
MPI_Isend(&b, 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &req[1]);
MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
}
else {
MPI_Recv(&c, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Recv(&d, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
printf("%c, %c \n", c, d);
}
...
```

Always prints a,b

```
#include<mpi.h>
int main() {
MPI_Init(NULL, NULL);
int myrank;
char a,b,c,d;
a='a'; b='b';
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank==0) {
MPI_Send(&a, 1, MPI_CHAR, 2, 0, MPI_COMM_WORLD);
}
else if (myrank==1){
MPI_Send(&b, 1, MPI_CHAR, 2, 0, MPI_COMM_WORLD);\
}
else if(myrank==2){
MPI_Recv(&c, 1, MPI_CHAR, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
MPI_Recv(&d, 1, MPI_CHAR, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
printf("%c, %c \n", c, d);
}
MPI_Finalize();
}
```

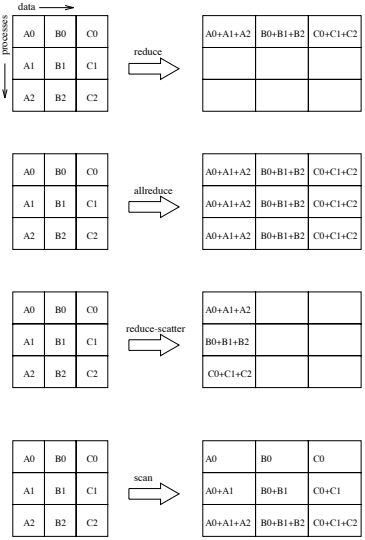
```
#include<mpi.h>
int main() {
MPI_Init(NULL, NULL);
int myrank;
char a,b,c,d;
a='a'; b='b';
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank==0) {
MPI_Send(&a, 1, MPI_CHAR, 2, 0, MPI_COMM_WORLD);
}
else if (myrank==1){
MPI_Send(&b, 1, MPI_CHAR, 2, 0, MPI_COMM_WORLD);\
}
else if(myrank==2){
MPI_Recv(&c, 1, MPI_CHAR, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
MPI_Recv(&d, 1, MPI_CHAR, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
printf("%c, %c \n", c, d);
}
MPI_Finalize();
}
```

Either prints a,b or prints b,a

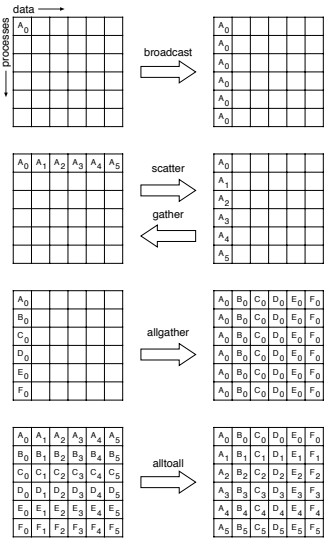
Collective communications

- Computation collectives: `Reduce`, `Allreduce`, `Reducescatter`
- Synchronization collective: `Barrier`
- Data movement collectives: `broadcast`, `scatter`, `gather`, `allgather`, `alltoall`

Computation collectives



Communication collectives



alltoall

`MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)`

Each process send to every process (itself included) the same amount of data

- `sendbuf` send buffer

alltoall

`MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)`

Each process send to every process (itself included) the same amount of data

- `sendbuf` send buffer

alltoall

`MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)`

Each process send to every process (itself included) the same amount of data

- `sendbuf` send buffer
- `sendcount` number of elements sent to each process

alltoall

`MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)`

Each process send to every process (itself included) the same amount of data

- `sendbuf` send buffer
- `sendcount` number of elements sent to each process
- `sendtype` datatype of send buffer elements

alltoall

`MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)`

Each process send to every process (itself included) the same amount of data

- `sendbuf` send buffer
- `sendcount` number of elements sent to each process
- `sendtype` datatype of send buffer elements
- `recvbuf` receive buffer

alltoall

`MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)`

Each process send to every process (itself included) the same amount of data

- `sendbuf` send buffer
- `sendcount` number of elements sent to each process
- `sendtype` datatype of send buffer elements
- `recvbuf` receive buffer
- `recvcount` number of elements received from any process

alltoall

`MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)`

Each process send to every process (itself included) the same amount of data

- `sendbuf` send buffer
- `sendcount` number of elements sent to each process
- `sendtype` datatype of send buffer elements
- `recvbuf` receive buffer
- `recvcount` number of elements received from any process
- `recvtype` data type of receive buffer elements

alltoall

`MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)`

Each process send to every process (itself included) the same amount of data

- `sendbuf` send buffer
- `sendcount` number of elements sent to each process
- `sendtype` datatype of send buffer elements
- `recvbuf` receive buffer
- `recvcount` number of elements received from any process
- `recvtype` data type of receive buffer elements
- `comm` communicator

alltoall

`MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)`

Each process send to every process (itself included) the same amount of data

- `sendbuf` send buffer
- `sendcount` number of elements sent to each process
- `sendtype` datatype of send buffer elements
- `recvbuf` receive buffer
- `recvcount` number of elements received from any process
- `recvtype` data type of receive buffer elements
- `comm` communicator
- `request` communication request

`MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, request)`

Each process sends to every process different amounts of data

- `sendbuf` send buffer
- `sendcounts` array with number of elements sent to each process
- `sdispls` array with displacement from `sendbuf` start of each sent block
- `sendtype` datatype of send buffer elements
- `recvbuf` receive buffer
- `recvcounts` array with number of elements received from each process
- `rdispls` array with displacement from `recvbuf` start of each received block
- `recvtype` data type of receive buffer elements
- `comm` communicator
- `request` communication request

Example: graph traversal with Alltoallv

First communicate size of messages, then communicate data
Reminder:

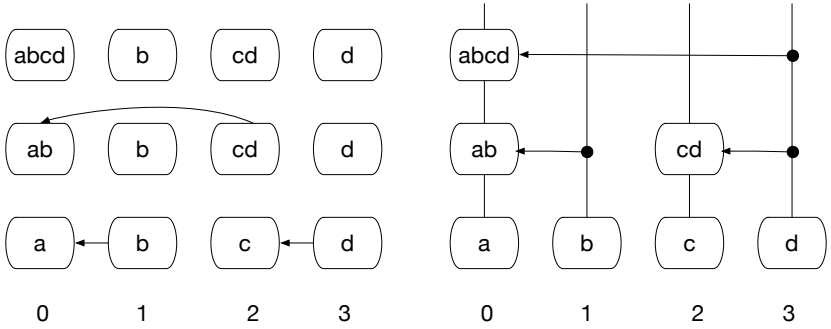
- `out[i][]` contains indices to be sent to process `i`. Each row has `max` elements
- `out_ptr[i]` contains number of indices to be sent to process `i`
- `in[]` is the receive buffer for all indices.

```
for(i=0;i<size;i++)
 sdispl[i] = rank*max;
out_ptr(rank) = 0;
    ...
MPI_Alltoall(out_ptr,1,MPI_INT,recvcounts,1,MPI_INT,MPI_COMM_WORLD)
rdipls[0]=0;
for (i=1; i<size; i++)
 rdispls[i]= rdispls[i-1]+recvcounts[i];
MPI_Ialltoallv(out,out_ptr,sdipls,MPI_INT,in,recvcounts,rdispls,MPI_INT,MPI_CO
...
```

How is reduce implemented?

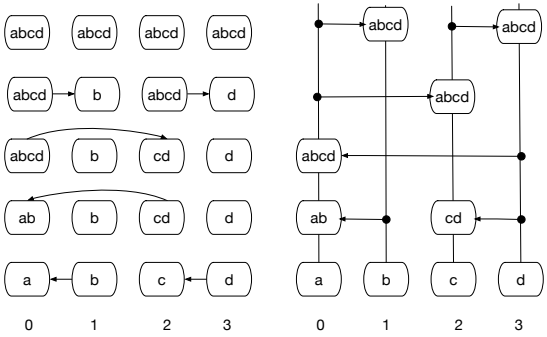
- Assume we reduce vector of length n
- Simple implementation: All processes send message to root; root sums them all.
- Assume receive of n words takes time $\ell + n/b$; only one receive at a time
- Communication time is $(p-1)(\ell + n/b) = \Theta(pn)$ – not good

Binary reduction



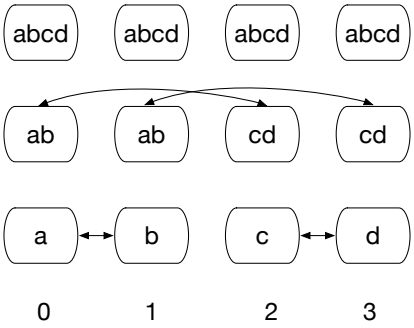
- Communication time is $\lg(p)(\ell + n/b) = \Theta(log(p) \cdot n)$ – much better
- Different processes exit collective operation at different times

Allreduce – reduce followed by broadcast



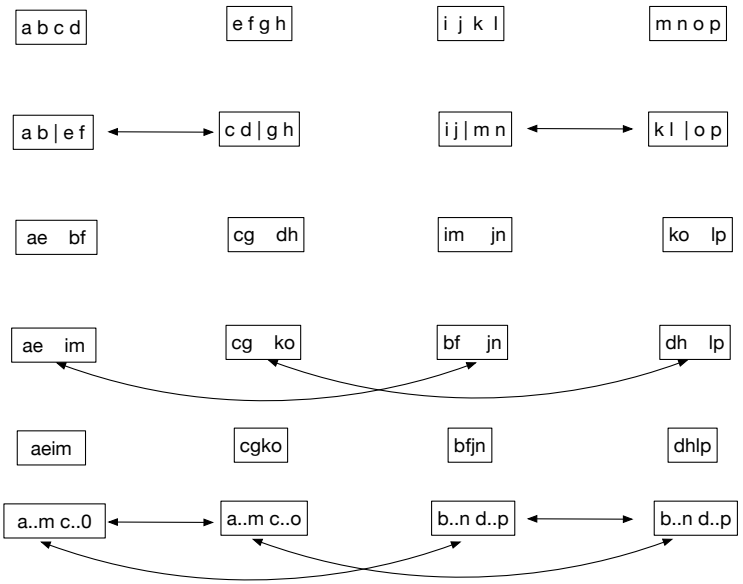
- Communication time is $2\lg(p)(\ell + n/b)$

Allreduce – alternative implementation



- Assume send and receive can be simultaneous: Communication time is $\lg(p)(\ell + n/b)$ – half as much as previous algorithm
- Assume no overlap at all between send and receive; then the two algorithms take as much time.

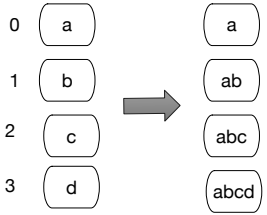
Allreduce – possibly better

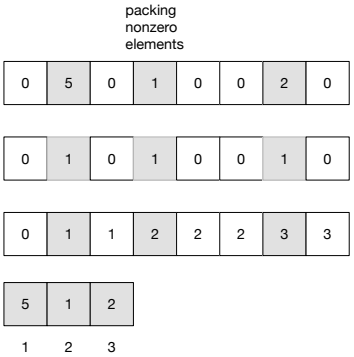


- Assume send and receive can be simultaneous
- Time is $2\lg(p)\ell + 2n/b$ – Better, for long vectors
- Optimal (up to constant factors)

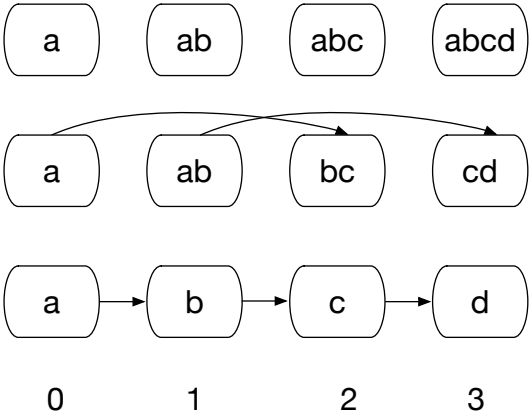
Scan

Examples





Scan evaluation



- Assuming simultaneous send and receive
- Time is $\lg(p)(\ell + n/b)$

Good MPI implementation uses *polyalgorithm*: It chooses the right algorithm, based on machine properties (latency, bandwidth), number of involved processes, length of messages, etc.